December 17, 1990

IBM Confidential

Annual Fellow Report for 1990
John Backus

The Functional Programming Project

Project Personnel

Alexander Aiken, Thom Linden, Peter Lucas, Paul Tucker (on loan from Menlo Park), John H. Williams, Edward L. Wimmers, John Backus

Introduction

In this report we will discuss the areas in which progress has been made in 1990. In this Introduction we present the highlights of our progress.

Highlights of progress and status in 1990

The language

As more and more programs are written in FL, we are finding the language to be powerful and convenient for a wide variety of tasks. The entire language is implemented in the current compiler. Provided that the optimization technology we are developing can be made to work as well across the board as it now does on our examples, our FL programming experience to date indicates that the FL language can serve as a very high level "fast prototyping" language (whose accuracy and robustness is enhanced by powerful type checking) and yet at the same time yield fast, production-level object code directly from the compiler. FL should thus eliminate the need to recode "fast prototype programs" in C while providing a more powerful and extensible language than those currently used for prototyping.

Type analysis

The task of type analysis is to determine what kinds (or *types*) of arguments each function can "see" in a program, and what kinds of results it can produce. The current type analysis system of the compiler (new this year) provides extremely detailed type information about an FL program. This information is vital input for the optimizer. It is also extremely helpful to the user in finding errors. Accurate type analysis is crucial for making programs correct, reliable and fast. No other type system for a functional language provides such precise type information. And unlike other languages, the FL type system does type inference without

the assistance of language restrictions that simplify type inference but makes programming more difficult (the usual case). The type system can prove, for example, that a particular use of a function (1) will always terminate (for a recursively defined function) or (2) will never produce an exception or (3) will always produce one of the values {1,2,3} or (4) will always produce a sequence of integers, etc..

The type analysis system is based on an approach involving several novel elements. Types are not sets of values, as in the standard approach, but sets of normal form expressions; this provides more precise information than the standard approach. The system uses a formal language for describing types that is based on the notion of "regular trees". Because FL is not "strongly typed" (i.e., is not restricted by type considerations), it is possible to write expressions whose types cannot be determined at compile time. For example, a function that computes the string odd when its argument is an odd integer and 1 when it is even may have a type (`integer` → `string` ∪ `integer`) that is determinable at compile time (even though such a function is not permitted in a strongly typed language). But in rare cases the type system may fail to determine the true type of a function (which might be `integer` → `integer`) and report its type as `integer` → (any type). If a use of this function required that its result be an integer, then this use would have to check at runtime that the result is an integer (and the compiler would create this check).

A paper [Aiken, Murphy 90] describing the FL type analysis system has been accepted for the 1991 Principles of Programming Languages Conference, the premier conference in this area.

Optimization

The optimizer, using rewrite rules derived from algebraic laws for the functions of FL and data from the type analysis system, has succeeded this year in performing some difficult and surprising transformations of programs. Its achievements give us reason to believe that the optimizing technology we are developing will be able to do the extremely demanding task of turning simple and clear but very inefficient FL programs into quite complex but very fast C programs – the difficult programs that today's programmers have to write themselves in languages like C.

The optimizations currently achieved far exceed in scope those done by existing optimizing compilers, such as those developed at Yorktown. They are important because, if achieved on most programs, this radical optimization technology will make possible a new style of programming that is far more efficient than is possible with current languages. Programs would become much cheaper and more reliable, and they would be produced much faster. "Fast prototyping" and programming would become one.

However, the results so far are just examples; we cannot be sure that we can achieve this level of optimization across all, or even most, programs. And these results are measured by looking at the resulting Lisp code; we have not yet produced the C code that would be needed to realize the running speed that these results make possible. (We need more manpower to do this.)

The most difficult and surprising optimization achieved by the optimizer so far is that of a bizarre FL matrix multiply program, one that is deliberately and extremely inefficient – using outer product – a program that copies matrices madly and takes time proportional to n^4 where n is the size of the matrices. The original program builds two sample matrices and then multiplies them together. The optimized program is the standard triply-nested loop for matrix multiplication (all the data copying, rearrangements and intermediate results of the original are gone), except that even the building of the sample matrix elements, as well as the multiplication itself, takes place in the inner loops, so that instead of two nests of loops, one to build the matrices and another to multiply them, there is just one.

The optimized program could only be improved upon by using deep mathematical results about matrix multiplication. Few programmers would have been able to do as well – including the correct construction of the matrix elements in the inner loops. The optimized program also produces the identical error indications (exceptions) as the *original* program, when it (the optimized program) is applied to inappropriate arguments. Preserving behavior in response to errors during program transformations is an important and difficult accomplishment – difficult because the original and transformed programs may be completely different in structure – important because error handling and debugging must take place in terms of the original program.

Other technical achievements

An important task of the simplifier section of the compiler is to remove variables from expressions that are introduced for readability. For example, it is easier to write `map(f)` = (some expression in `f`), than it is to write the higher order functional expression for `map` directly. The process of turning a (readable) expression containing variables into (a less readable) one with no variables is called *abstraction.* Unfortunately, all the standard abstraction algorithms do at least one of the following things, all of which pose serious problems for optimization:

1. Produce expressions that are very much larger than their input expressions
2. Produce complex, unnatural expressions that are unlike the original one
3. Introduce new combining forms.

A new abstraction algorithm was designed and incorporated in the simplifier during 1990. It produces expressions of about the same size as the original and with the same structure, and introduces no new combining forms. In addition to the theoretical interest of this work, the new algorithm means that (a) abstraction does not produce huge, difficult-to-optimize expressions and (b) our optimization strategies for normal FL expressions now apply equally well to expressions resulting from abstraction. A paper is expected on this work.

Progress on the components of the compiler

This section presents the details of the work on the compiler during 1990.

The parsers and lexical scanners

The language parser and lexical scanner turns an FL program into a tree of R-nodes, the internal form used throughout the compiler. It does a great amount of simplification and elimination of various language constructs. The rules parser and scanner translate rewrite rules in text form into the internal tree form required for the rewrite engine.

During 1990 both parsers and scanners were frequently modified to implement language features that had not been implemented (e.g., libraries, includes) and to produce new kinds of R-node features that were found necessary as serious rewriting began to indicate needs for new features. This work was done by Peter Lucas during the first half of 1990 and by Thom Linden during the second half.

The simplifier

The simplifier's task is to transform the tree form program produced by the parser into a single set of global function definitions in which the right hand side of each definition is built using only application and sequence formation. The parser has done everything that is feasible to do in a bottom-up fashion. The simplifier has to deal with problems that require top-down analysis; these problems involve interactions and flow of information between widely separated entities. This makes it very difficult for the simplifier to do all the work in two passes (which it does).

The main tasks of the simplifier first pass are (1) to produce a single global environment that contains all the function definitions that occurred in any of the local environments, while creating a flattened name space to resolve conflicting names and making sure that the right sides of definitions are function-valued expressions, (2) put expressions into normal form (without unnecessary lifting), (3) produce a table of the transitive closure of all function calls (this reveals all the recursively defined functions). The second pass of the simplifier abstracts each expression with respect to all of its variables simultaneously, producing a variable-free expression.

The simplifier that existed at the beginning of the year was a one-pass simplifier but it had a serious flaw: the elimination of lambda expressions with multiple lambdas often resulted in an exponential blowup in the size of the resulting lambda-free expression. It also produced expressions that were so convoluted and unnatural that they were difficult to optimize.

The new simplifier uses a linear algorithm for removing lambda abstractions (in the worst case it is O(nlog(n)), but these cases do not occur in practice). (A linear abstraction algorithm takes time that is proportional to the size of the expression.) There are several linear abstraction algorithms in the literature, but they tend to completely scramble the

expressions they work on. Another technique, the use of "supercombinators", introduces *new* combining forms (high level functions for building programs).

The FL problem is to produce variable-free expressions that use the *standard* FL combining forms and are close to the kinds of expressions people write. This allows us to invent reasonable rules to transform these expressions, rules that are based on the algebraic properties of the FL combining forms. None of the abstraction techniques in the literature satisfy these needs. This work has been done by John Williams and Ed Wimmers; they expect to write a paper about it.

0.1 The type analysis system

POPL paper

The type analysis system analyzes an FL program to determine, for each use of a function, what types of arguments it can get and what types of results it can produce. The implementation of the first version of the type system, begun in late November 1989, was running in early March 1990. This version had serious performance bottlenecks. The current version was implemented from March to May.

The current version is capable of analyzing programs at roughly a rate of 1-3 lines of FL per second on an RT. While this is not fast, it is acceptable performance for a research compiler. This level of performance was difficult to achieve; several of the algorithms and heuristics that were needed involved significant theoretical advances. For comparison, it is believed that the current system is as fast, and probably much faster, than other program analysis systems for functional languages. The type system currently consists of approximately 14,000 lines of Common Lisp code. Virtually all of this has been written, or rewritten, by Alex Aiken.

In other functional programming systems type information is not used for extensive optimization of functional programs; as a result, the FL type system differs widely from traditional type systems. The most important differences are: (1) it is based upon an *abstract interpretation* of the operational semantics of FL, and (2) *regular trees*, a kind of finite automata, are used to describe types. Experience with the type system has demonstrated that type information is a crucial ingredient in the optimization of FL programs. It appears that almost any non-trivial optimization requires type information.

The precision of the type system makes it useful for more than program optimization. The type system routinely detects and reports both common and subtle programming errors. The type system can answer questions such as: "What are the possible errors that can occur in function *f* if the argument is a nonnegative integer?" This capability and the precision of the system suggests the possibility of a new and powerful debugging tool that would allow a user to ask questions about the behavior of programs.

A paper, "Static Type Inference in a Dynamically Typed Language" by Alex Aiken and Brian Murphy (an MIT Co-op student who worked with Aiken for six months in 1989) has been accepted for presentation at the 1991 POPL Conference.

The rewrite system

The code generator

The graphics interface

Primitives

General remarks on our status

Miscellaneous activities (John Backus only):